Basic Breakdown of GoLab

GoLab is a collaborative code editing application that allows multiple users to create, edit, and run Go code. The system will resolve inconsistencies from concurrent edits using CRDTs and achieve load balancing using consistent hashing.

The system consists of 4 types of nodes: users, editor nodes, file system nodes, and a single server.

There would need to be an API exposed for the **user** applications to communicate with the **editor** nodes. A protocol for **editor** nodes to communicate together, and a protocol for **file system** nodes to do the same.

A **user** is a simple web application controlled by a physical person. The application presents a user interface granting the ability to read, write, and execute Go files within a distributed file system.

A new **user** first contacts a central server which assigns an **editor** node to the user, which the user implicitly communicates with by means of interactions with the UI (modifying source, running file, etc).

Files changes can be observed by a user making modifications to a file, or by back-end logic that maintains file state. After a **user** makes some file modifications, a **commit** will be issued to their **editor,** which will send this **commit** to the **file system.** The **editor** node will run the file to make sure the code compiles without errors.

The **server** acts as a load balancer and maintains the topology of both the network of editor nodes as well as nodes in the distributed file system. Given each **editor** is connected to some other number of **editors**, each node must flood file changes to their respective nodes. The **editor** nodes will use sequence CRDTs to handle concurrency.

(***Diversion:*** It is to be assumed that before any **users** begin writing source code, there are test cases written that ensure the expected execution results of the code base. In this way, this is TDD enforced system.)

At any point, a user may decide to run the code in the current file open for editing. They do so by creating a *branch* - a “copy” of the project editable only by the user issuing the run command. After the branch is created, the user can make last-minute adjustments without interference from other users to make sure the code compiles before running. When a run command is issued, the editor node creates a local copy of the project, makes any adjustments needed to match with the newly created branch, and downloading all other necessary files from the DFS and then executes the code.

After a successful run on the editor node, a *commit* can be created. A **commit** is some working copy of the project with changes over top (a set of file deltas, so to speak). A project consists of a set of files as well as all the **commits** on those files on top of the last **voted** working copy of the project. Since it is expected that code is being written to satisfy some test suite, the nodes in the system must work together by running tests, checking for compilation, verifying users, etc. to decide which **commit** will be the new working copy. Once this is done, the **file system** must flood to the **editor** network, which will subsequently update their local states and that of their respective **users.**

The **file system nodes** contain primaries and replicas of working copies and is responsible to maintain file version history.

Fault Tolerance:

- If an **editor** node fails with active user applications, the user applications will contact the **server** to connect to a new **editor** node.

The connected **editor** nodes to the failed one would need to contact the **server** for new connections if below a certain threshold